Program Synthesis from Natural Language Using Recurrent Neural Networks

Presented by Dewi Yokelson, April 2019

Paper by Xi Victoria Lin & Others

GOAL

- Make programming easier and more productive by letting programmers use their own words and concepts to express the intended operation
- Avoid wasted time searching online when the programmer does not know key words to search or cannot find the answer
- Man pages can be hard to discover and understand

Natural Language to bash

Question 1. I have a bunch of ".zip" files in several directories "dir1/dir2", "dir3", "dir4/dir5". How would I move them all to a common base folder? (http://unix.stackexchange.com/questions/67503)

Solution: `find dir*/ -type f -name "*.zip" -exec mv {} "basedir" \;`

Question 2. I have one folder for log with 7 sub-folders. I want to delete all the files older than 15 days in all folders including sub-folders without touching folder structure. (http://unix.stackexchange.com/questions/155184)

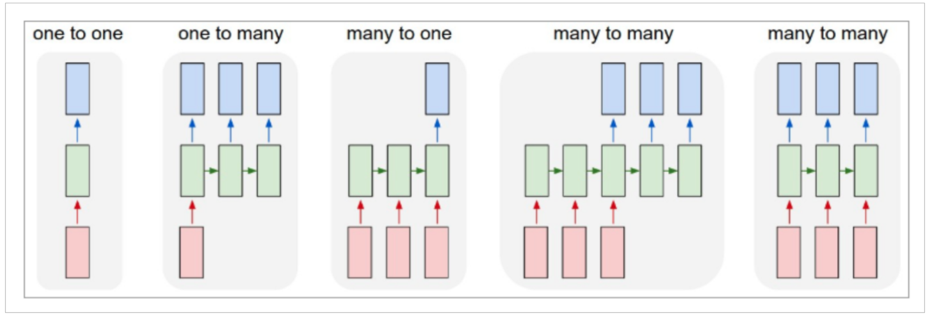
Solution: `find . -type f -mtime +15 | xargs rm -f`

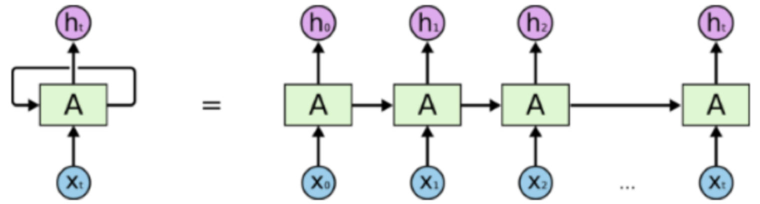
Tellina

- Does the translation using recurrent neural networks (RNNs)
- An interactive web page where you type in your natural language statement and you receive a ranked list of possible bash one line commands
- http://tellina.rocks

Recurrent Neural Networks

- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.



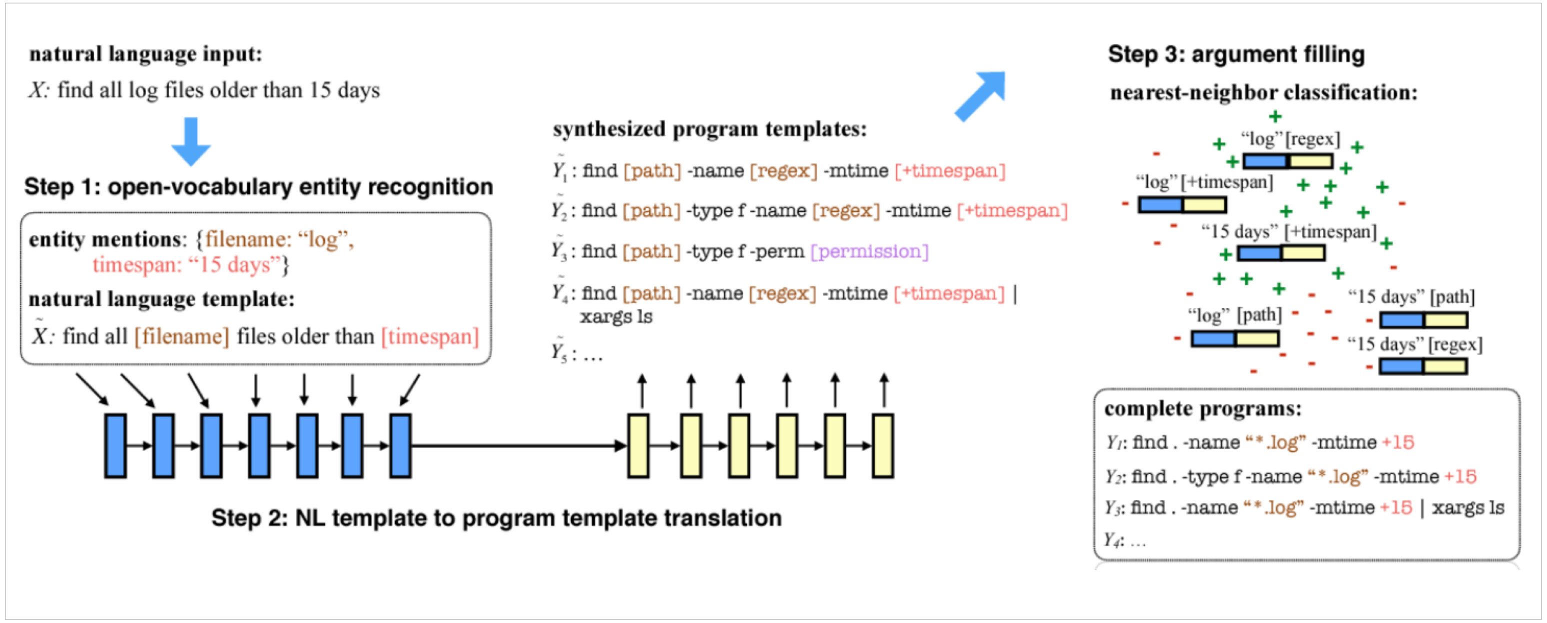


An unrolled recurrent neural network.

- Traditional neural nets accept fixed sized vectors as input and produce fixed sized output, not so with RNNs

The Approach

- User provides natural language sentence X which Tellina tranforms into a template
- An RNN encoder-decoder model translates the template into a ranked list of possible program templates with argument slots
- The argument slots are replaced by program literals to produce an output program using a k-nearest neighbor classifier



Template Generation

- Pattern
  - *File*: file name
  - *Directory*: directory name
  - *Path*: absolute path
  - *Permission*: Linux file permission code
  - *Date/Time*: date and time expression
  - *Regex*: other pattern arguments
- Quantity
  - *Number*: number
  - *Size*: file size
  - *Timespan*: time duration

- Used a domain-specific heuristic: defined two categories of entities, *patterns* and *quantities*
- To recognize and assign types to natural language commands they manually defined regexs and mapped them to their type
- To recognize and assign types in the bash command templates they map man page types to the types above

Global entity-slot alignment

Algorithm 1: Global entity-slot alignment

Input : List of entities E, list of argument slots S, local entity-slot compatibility function $\gamma(i,j)$.

Output: List of matched entity-slot pairs M if every entity is aligned to a slot; `null` otherwise.

$M = \emptyset$;
/* *compute the preference list for each entity* */
for $e_i \in E$ **do**
    PriorityQueue S_{e_i};
    for $s_j \in S$ **do**
        if $\gamma(i,j) \neq$-*inf* **then**
            S_{e_i}.Enqueue(s_j)
        end
    end
end
/* *compute the stable alignment* */
while $\exists e_i$ *s.t.* $\forall s_j (e_i, s_j) \notin M \wedge S_{e_i} \neq \emptyset$ **do**
    $s_j = S_{e_i}$.Dequeue();
    if $\exists e_{i'}$ *s.t.* $(e_{i'}, s_j) \in M$ **then**
        if $\gamma(i',j) < \gamma(i,j)$ **then**
            $M = M \cup \{(e_i, s_j)\} \setminus \{(e_{i'}, s_j)\}$;
        end
    else
        $M = M \cup \{(e_i, s_j)\}$;
    end
end

Program Slot Filling

- Often a one to one mapping between the entities in NL and the resulting program
- Tellina aligns the most likely entities using the Global Entity-Slot Alignment algorithm (previous slide) and then extracts the values from the NL sentence and inserts them into the program
- Each entity-slot pair (e_i ,s_j) is represented using the concatenation of the hidden state vectors (h_i ,h'_j) of the neural encoder-decoder model

$$\gamma(i,j) = \sum_{(c,d) \in NN(i,j,k)} d_{(i,j),(c,d)} \cdot v(c,d),$$

$$d_{(i,j),(c,d)} = \cos((\mathbf{h}_i, \mathbf{h}'_j), (\mathbf{h}_c, \mathbf{h}'_d)),$$

$$v(c,d) = \begin{cases} 1, \text{if } (e_c, s_d) \text{ match} \\ 0, \text{otherwise} \end{cases}.$$

Data

- Labor-intensive data collection process: hired workers to scrape the web for ultimately just over 5000 nl-bash pairs

In-scope syntax structures:

- Single command
- Logical connectives: `&&`, `||`, parentheses `()`
- Nested commands: pipeline `|`, command substitution `$()`, process substitution `<()`

Out-of-scope syntax structures:

- I/O redirection `<`, `<<`
- Variable assignment `=`
- Parameters `$1`
- Compound statements: `if`, `for`, `while`, `until`, blocks, function definition
- Regex structure (every string is a single opaque token)
- Non-bash program strings triggered by command interpreters such as `awk`, `sed`, `python`, `java`

Evaluation

Model	Acc_F^1	Acc_F^3	Acc_T^1	Acc_T^3
CR Baseline	13.0%	20.6%	54.7%	67.9%
Tellina Model	30.0%	36.0%	69.4%	80.0%

Table 2: Translation accuracies of the Tellina model and the code retrieval baseline.

k	Precision	Recall	F1
1	82.9	87.0	84.9
5	84.6	89.0	86.7
10	82.1	86.2	84.1
100	79.8	84.0	81.9
200	77.2	81.2	79.1

Table 3: Development set performance of the argument filling component for differing k nearest neighbor values.

User Study

- Conducted a user study to determine whether Tellina helps programmers complete file system tasks using bash
- Recruited 39 CS students, all familiar with bash
- Assigned 2 tasksets made up of 8 tasks, for each taskset they were either allowed to use Tellina or not
- Overall success rate 88%, participants using Tellina on average used 22% less time and had a 90% success rate over the 85% in the control group (without Tellina)

Questions?